Loft+Cyclone

Frédéric Boussinot

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Loft+Cyclone

Frédéric Boussinot

N° 5680

Septembre 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5680--FR+ENG

INRIA
SOPHIA ANTIPOLIS

Loft+Cyclone

Frédéric Boussinot*

Thème COM — Systèmes communicants
Projet Mimosa

Rapport de recherche n° 5680 — Septembre 2005 — 16 pages

Abstract: This paper reports on an experiment to add concurrency to the Cyclone programming language, in order to get a safe concurrent language. The basic model considered is that of FairThreads in which synchronous and asynchronous aspects are mixed. The language Loft implements the FairThreads model in C. In this paper, one uses Cyclone instead of C in the implementation of Loft. Using the multi-threaded version of Boehm's GC, one gets an extension of Cyclone to concurrency which is as safe as Cyclone for sequential code, with some additional safety verifications for concurrent code.

Key-words: Reactive programming, Safe programming, Cyclone

* with support from ACI Securité Informatique, ALIDECS project

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Loft+Cyclone

Résumé : Ce rapport décrit une expérience d'ajout de concurrence au langage de programmation Cyclone, avec pour objectif la conception d'un langage concurrent sûr. Le modèle de départ est celui des FairThreads qui combine des aspects synchrones et asynchrones. Le langage Loft implémente le modèle des FairThreads en C. Dans ce rapport, on remplace C par Cyclone dans l'implémentation de Loft. En utilisant la version multi-threadée du GC de Boehm, on a obtenu une extension Concurrente de Cyclone, aussi sûre que celui-ci en ce qui concerne le code séquentiel, avec des possibilités de verification supplémentaires pour le code concurrent.

Mots-clés : Programmation réactive, Programmation sûre, Cyclone

1 Introduction

This paper reports on an experiment to add concurrency to the Cyclone[6] programming language. Cyclone is safe: no crash can occur during program execution. However, Cyclone is presently limited to sequential programming. The objective of the present work is to get a safe concurrent language.

The basic model is that of FairThreads[3] in which synchronous and asynchronous aspects are mixed. In this model, several schedulers are running asynchronously. Each scheduler defines a synchronous area in which threads linked to the scheduler are executed in a cooperative way, sharing the same instants of execution. A thread can migrate from a scheduler to another one during execution. A thread can unlink from a scheduler to run some sequential code asynchronously; typically, a thread unlinks from a scheduler for the time needed to perform a blocking system call (a `read`, for example). Thus, while blocked, the unlinked thread does not prevent the others threads linked to the scheduler to execute.

The language Loft[1] implements the FairThreads model in C. In this paper, one uses Cyclone instead of C. The simple replacement of C by Cyclone is not however sufficient to get a fully safe concurrent language. In particular, one has to ensure that an error occurring in a thread will not propagate to others threads, preventing them to execute.

The paper first introduces the Loft+Cyclone language. Then, safe concurrent programming is presented and discussed. Related work is finally considered before conclusion.

2 Overview of the language

Basic units of concurrency are *threads* which are created as instances of *modules*. Modules are introduced by the keyword `module` and terminated by `end module`. Modules have a name and a new thread instance of the module named `m` can be created with the function `m_create`. There is an *implicit scheduler* to which created threads are automatically linked by default (if no other scheduler is specified). Modules and Cyclone code can be intermixed. For example, consider:

```
#include <stdio.h>
typedef const char ? str;

module print_loop (str msg)
while (1) do
   printf ("%s",local(msg));
   cooperate;
end
end module

module main ()
print_loop_create ("hello ");
print_loop_create ("world ");
end module
```

Two modules are defined, the first one named `print_loop` and the second one named `main`. The first module calls the cyclone function `printf` which needs the inclusion of the file `stdio.h` (line 1). The second line defines a Cyclone type named `str`.

Module `print_loop` has a parameter of type `str`, and its body is made of a `while` infinite loop which, at each instant, prints the parameter and cooperates. Note the loop syntax `while (...) do ... end` which differs from the standard C syntax. Note also that the parameter is accessed through the macro `local`; this is the rule in Loft where variables which are local to a thread (parameters belong to these) are always accessed in this way.

The `main` module creates two threads instances of `print_loop`. Because of its name, an instance of the module `main` is automatically created and run by the implicit scheduler.

Here are the commands to compile the program (let us suppose it is contained in the file `hello.loft`):

```
loft2cyc < hello.loft > hello.loft.cyc
cyclone $(FLAGS) hello.loft.cyc
              print_loop_to_automaton.c _main_to_automaton.c
              $(LIBS)
```

First, Cyclone code is produced with the `loft2cyc` command. Then, the produced code is compiled with the Cyclone compiler (the two variables `FLAGS` and `LIBS` are supposed to be correctly set). Note that two auxiliary C files have been also produced by `loft2cyc`, one for each module; these C files have to be compiled and linked with the Cyclone code.

The program is run with the command `a.out`. Execution prints "`hello world`" forever. An important point is that the order of the two messages always remains the same: the program is actually totally deterministic. The two threads created by running the instance of `main` are linked to the implicit scheduler. The implicit scheduler runs the first thread (which prints "`hello`"), then the second thread which prints "`world`". At that point, all the threads linked to the implicit scheduler have cooperated and the global instant is over. Execution of the next instant can start, and the two linked threads are executed again, in the same order (the order in which the threads have been linked to the scheduler). Actually, the same execution takes place at each instant.

2.1 Atoms

In the module `print_loop`, the call of the function `printf` of Cyclone is called an *atom*. Actually, atoms are either function calls or blocks of Cyclone statements of the form "`{...}`". In both cases, atoms are run by the executing thread in one single step and the execution is intended to complete at the same instant it is started. One says that execution of atoms *takes no time*.

It is the programmer's responsibility to guarantee termination of atoms. For example, let us consider the atom:

```
{ while (1)/* nothing */; }
```

A linked thread executing the atom would never complete it and thus never cooperate. Thus, execution would stay in it for ever (recall that we are not in a preemptive context, but in a cooperative one) which would prevent the others threads linked to the same scheduler to execute. Actually, in such a situation, the current instant would never terminate. Note however that there is no problem if the executing thread is unlinked instead of being linked to a scheduler.

The programmer has another responsibility: atoms should not take too much time to execute, to obtain a correct global responsiveness. This rather informal obligation, more diffuse than atom termination, of course depends on the application context.

2.2 Events

The language introduces *events* as a specific means for threads to synchronize and to communicate. An event is a user defined entity which is associated to a given scheduler (as for threads, the implicit scheduler is the default) and which, at each instant, is either present or absent. At the beginning of each instant, all events are automatically reset to absent. The important point is that the presence/absence value of an event is *always coherent* for all the threads linked to the event scheduler. It is guaranteed that, during the same instant, all the threads "see" the same presence/absence of an event; in short, *events are broadcast*. Thus, it is not possible for a thread to consider an event as absent while another thread considers it as present during the same instant, independently of the way the two threads are scheduled.

Events are of the type `event_t` and the basic associated instructions are `generate` to set an event as present for the current instant, and `await` to wait for an event to be present.

As an example, let us implement condition variables (in the spirit of the POSIX ones). A condition variable is basically an integer variable (initialized to 0) with an associated event to signal assignments to it:

```
int cond = 0;
event_t cond_sig;
```

To set the condition, one writes the atom:

```
{
   cond = ...;
   generate (cond_sig);
}
```

Here is the code to await for the condition (`exit_loop` is a local variable used to exit the loop):

```
{local(exit_loop) = 0;}
while (!local(exit_loop)) do
   await (cond_sig);
   if (cond == ...) then
      {local(exit_loop) = 1;}
```

```
   else
       cooperate;
   end
end
```

Because an event is used, threads waiting for the condition to be fulfilled do not have to poll it, but can just sleep while the condition remains unchanged. Because `cond_sig` is broadcast, all the threads waiting for the condition are awaken as soon as the condition is changed. However, there is no guarantee that the same value is read (if for example two threads set it with different values); additional code is needed to let this property hold.

An important point is that there is no need of any locking mechanism to ensure correct access to the condition variable; this is a consequence of the cooperative approach used: there is no risk of preemption while setting or reading the condition variable.

2.3 Valued events

Valued events are events to which values can be associated when they are generated. These values are broadcast. Valued events are of the type `event_value_t<'a>`, and the values should be of the pointer type `<'a>*`. The function `generate_value` is used to generate a valued event with an associated value. As a simple event, a valued event can be awaited. The instruction `get_all_values` receives all the values generated for an event during the current instant, and calls a call-back function on each of them. As all the values are collected, the execution of `get_all_values` lasts during the whole instant, and terminates only at the next instant.

To illustrate the use of valued events, let us implement synchronization barriers. A synchronization barrier blocks the threads reaching it while their number is less than a given threshold; when the threshold is reached, the barrier lets the blocked threads resume their execution. The arrivals to the barrier are counted using a valued event (with `void*` values):

```
event_value_t<void> arrival;
event_t go;
```

To notify arrival at the barrier, a thread generates `arrival` with the value `NULL`, and awaits `go` to continue:

```
generate_value (arrival,NULL);
await (go);
```

The barrier counts the number of arrivals and broadcasts `go` as soon as the threshold is reached (the function `count_callback` just increments its argument):

```
{
   local(exit_loop) = 0;
   local(count) = 0;
```

```
}
while (!local(exit_loop)) do
   await (reach);
   get_all_values (arrival,count_callback,&local(count));
   if (local(count) >= threshold) then
      {local(exit_loop) = 1;}
      generate (go);
      cooperate;
   end
end
```

The new arrivals are not counted during the instant where `go` is generated as the `cooperate` instruction blocks execution of the barrier for this instant. The threads reaching the barrier at this instant proceed immediately, because `go` is present.

2.4 Control over threads

Several means are defined to control a thread linked to a scheduler. More precisely, one can give the order to the scheduler to which the thread is linked to stop it definitively, or to suspend or to resume it. For example, let us consider an application which starts a service and then waits for its termination to exit[1]:

```
thread_t application, service;
event_t service_start, service_terminated;

module application ()
generate (service_start);
await (service_terminated);
exit (0);
end module
```

A normal service is modeled by a loop which prints 1 forever. It is defined in the module `service`. A `finalize` part is introduced in the module; it is executed when the service is killed by a stop order issued on it; in this case, the event `service_terminated` is generated:

```
module service ()
await (service_start);
while (1) do
   printf ("1");
   cooperate;
end
finalize generate (service_terminated);
end module
```

[1] Modules have a specific name space; thus a module and a variable or a function can share the same name.

Let us suppose that one has to replace the normal service by a new one called `supply` which prints 2 only a limited number of times and then terminates. The new service is:

```
module supply ()
await (service_start);
repeat (LIMIT) do
   printf ("2");
   cooperate;
end
generate (service_terminated);
finalize generate (service_terminated);
end module
```

In order to replace `service` by `supply`, one has to run an instance of the module `switch_service`. First, it suspends the application and stops the normal service. Then, at the next instant, it creates the supply service and resumes the application:

```
module switch_service ()
{
   suspend (application);
   stop (service);
   supply_create ();
}
cooperate;
{
   generate (service_start);
   resume (application);
}
end module
```

The suspension of the application is needed to mask the termination of the normal service when it is stopped. Note that the switch of services remains totally transparent for the application. Note also that "there is no loss of service": the supply service is executed as soon as the normal one disappears.

2.5 Linking and unlinking

A thread can unlink from a scheduler in order to run in an autonomous way, until it possibly re-links to the initial scheduler or to another one. During the time it is autonomous, the thread is only under the control of the OS. As it does not have any access to instants, an unlinked thread cannot wait for an event or get values from it. Only threads instances of a *native module* can unlink. A module is native if the `native` keyword immediately follows `module` in the definition.

A typical context of native module use is when blocking I/Os are needed. For example, let us suppose that one wants to print strings typed on the keyboard, using the blocking I/O function `getchar` which reads characters one by one.

One defines the function `getstring` which reads characters until a carriage return and fills the variable `global` with them. Variable `len` contains the number of characters read (for simplicity, the definition of `getstring` is omitted).

```
int ? global; int len = 0;
```

A native module is defined which calls `getstring` while unlinked, and then re-links to signal the printer module with the event `done`:

```
event_t done;
module native getstring_module ()
while (1) do
   unlink;
   getstring ();
   link (implicit_scheduler ());
   generate (done);
end
end module
```

The printer module awaits the event `done` and then prints the variable `global`, using `len` to get its length:

```
module print ()
while (1) do
   await (done);
   { for (int i = 0; i < len; i++) printf ("%c",global[i]); }
   cooperate;
end
end module
```

Note that the re-linking of `getstring_module` prevents the read string to be erased before being printed.

2.6 Using several schedulers

Several schedulers can be used in the same application. A scheduler is created with the function `scheduler_create` and it is run autonomously with the function `scheduler_go` (actually, each scheduler is mapped on a native thread).

For example, in the following code, two threads are created in two distinct schedulers which are run autonomously, and the output is a nondeterministic sequence of 0 and 1:

```
module trace (int i)
while (1) do
   printf ("%d",local(i));
   cooperate;
end
end module
```

```
module main ()
{
   let s1 = scheduler_create ();
   let s2 = scheduler_create ();
   trace_create_in (s1,0);
   trace_create_in (s2,1);
   scheduler_go (s1);
   scheduler_go (s2);
}
end module
```

Of course, the same nondeterministic output could as well be obtained using two unlinked threads.

The presence of several schedulers can help to achieve reactivity. For example, let us consider a system which has to execute heavy tasks and light tasks. Execution of heavy tasks needs more computing power than execution of light ones. If the focus is put on the reactivity of light tasks, one can define two autonomous schedulers, one for each kind of task. Compared to a solution with only one scheduler, execution of light tasks is not penalized by execution of heavy ones. The gain can be important when there are few light tasks compared to heavy ones. For example, let us model a heavy task as needing 10 instants to perform an output, and a light one as needing only one:

```
module heavy ()
while (1) do
   repeat (10) do cooperate; end
   printf ("h");
end
end module

module light ()
while (1) do
   printf ("l");
   cooperate;
end
end module
```

Let us suppose that there are 10 light tasks and 100 heavy tasks:

```
module main ()
{
   let light_sched = scheduler_create ();
   let heavy_sched = scheduler_create ();
   for (int i = 0; i < 10; i++) light_create_in (light_sched);
   for (int i = 0; i < 100; i++) heavy_create_in (heavy_sched);
   scheduler_go (light_sched);
```

```
   scheduler_go (heavy_sched);
}
end module
```

One measures the time needed for the light tasks to perform a fixed number of outputs. On a single processor machine, the solution with 2 schedulers is almost twice as fast as the solution with only one scheduler; this clearly indicates a better reactivity of light tasks.

The function `scheduler_instant` executes only one instant of a scheduler. Using it, one can pilot several schedulers, and for example, build priority systems, in which each scheduler has a priority and runs threads with same priority. For example, the following fragment of code defines two schedulers with fixed priorities, the one with higher priority running `HIGH` times more often (`HIGH` > 1) than the one with lower priority:

```
module main ()
local scheduler_t high, scheduler_t low;
   {
      local(high) = scheduler_create ();
      local(low) = scheduler_create ();
   }
   ...
   while (1) do
      repeat (HIGH) do
         scheduler_instant (local(high));
         cooperate;
      end
      scheduler_instant (local(low));
      cooperate;
   end
end module
```

There also exist primitives to control execution of schedulers (not considered here, for simplicity); for example, to let all threads linked to a scheduler execute once in turn, or to test if some thread needs to be continued in the current instant, or to start or terminate an instant. With these primitives, one can for example synchronize execution of autonomous schedulers. These primitives have been used in a multi-threaded implementation of cellular automata[4].

2.7 Multi-processing

Unlinked threads and autonomous schedulers are mapped on native threads executed in a preemptive way by the OS. Execution of these native threads can immediately benefit from multi-processor architectures, in particular SMP ones. However, as usual with preemptive threads, data races can occur if the same shared memory area is accessed concurrently by threads which are unlinked or which are linked to distinct autonomous schedulers. Actually, the model of FairThreads suggests a software architecture with the following characteristics:

- The shared memory should be partitioned in several areas, each area being mapped to a unique scheduler and only accessed by the threads linked to it. In this approach, before accessing an area of the shared memory, a thread has to link to the corresponding scheduler.
- Unlinked threads should only be allowed to access private memory; private memory should be initialized and its content should be copied into the shared memory only when the thread is linked to the appropriate schedulers.

In such an approach, schedulers can be seen as "big" locks which, just like standard locks in preemptive programming, protect shared memory from concurrent accesses. There is however a big difference: indeed, now, several threads can correctly and simultaneously (in the same instant) access the same shared memory, which is of course impossible with standard locks.

3 Safe concurrent programming

Safe programming roughly means absence of run-time errors leading to crashes. Actually, run-time errors can still occur in a safe programming context, but only under the form of exceptions that can be trapped by programmers.

The notion of safe programming needs to be extended to concurrency: a run-time error of one thread should never "propagate" and prevent the execution of the other threads. In Loft+Cyclone, this means that the following objectives are to be met:

1. A thread linked to a scheduler which raises an untrapped exception should be removed from the scheduler, without preventing the others threads to run.
2. No instantaneous loops can be run by a linked thread. Indeed, in this case, the others threads would never get the control.
3. No non-terminating atoms can be run by a linked thread. As in the previous case, the thread running a non-terminating atom would prevent the other threads from executing.

3.1 Exception handling

The following run-time errors are specific to the language and produce exceptions:

- Order to stop, suspend or resume an unlinked thread. Of course, this error can only appear for instances of native modules, the only ones that can be unlinked. In this case, it is good practice to enclose the order in a `try-catch` instruction.
- Waiting for an event or attempting to get values of an event not defined in the same scheduler.

These exceptions, along with the ones produced by Cyclone execution of atoms, are automatically captured by the executing thread, at the outermost level. When such an exception occurs at that level, the thread terminates definitively. If it was linked to a scheduler, it is consequently removed from it. Thus, an error occurring in one thread never propagates to the other threads.

3.2 Static analysis

Two static analyses are defined to be performed at compile time: the first one ensures that, when unlinked, a thread cannot execute primitives only defined for linked threads (for example, `cooperate` or `await`). The second analysis ensures that there is no instantaneous loop. As usual, these static analysis are approximate (conservative): there exist correct programs which are rejected.

Verification of correct linking

One defines a function V_{link} which analyses an instruction in a given context, and which either raises an error or returns the new context obtained after the analysis. A context actually reduces to a boolean: false means that the instruction is certainly linked and true that it is possible for it to be unlinked. V_{link} is defined by:

1. $V_{link}(link, b) = false$, and $V_{link}(unlink, b) = true$.
2. $V_{link}(i, b) = b$ for all instructions i which can execute both in a linked or in an unlinked context; this is for example the case of atoms.
3. $V_{link}(i, true)$ raises an error for all instructions i which should not be executed when unlinked, as `cooperate`. These instructions can only execute in a linked context; for them: $V_{link}(i, false) = false$;
4. $V_{link}(i_1; i_2, b) = V_{link}(i_2, V_{link}(i_1, b))$
5. $V_{link}(if(e)\ then\ i_1\ else\ i_2, b) = V_{link}(i_1, b)\ or\ V_{link}(i_2, b)$
6. $V_{link}(while(e)\ do\ i, b) = V_{link}(i, b)\ or\ b$

Initially, the program is considered to be linked (context false). The approximate nature of the analysis results from the use of *or* in the two last points, and from the fact that dead code is analyzed (point 4).

Detection of instantaneous loops

The function V_{loop} detects instantaneous loops; it takes an instruction as parameter and returns a boolean which is true if the instruction can terminate instantaneously. It is defined by:

1. $V_{loop}(i) = false$ for all instructions i that never terminate instantaneously, as `cooperate`, `get_all_values`, `link`, and `unlink`.
2. $V_{loop}(i) = true$ for all instruction that can terminate instantly, as the atoms.
3. $V_{loop}(i_1; i_2) =$ let $b = V_{loop}(i_1)$ in if b then $V_{loop}(i_2)$ else b
4. $V_{loop}(if(e)\ then\ i_1\ else\ i_2) = V_{loop}(i_1)\ or\ V_{loop}(i_2)$
5. $V_{loop}(while(e)\ do\ i)$ raises an error if $V_{loop}(i) = true$; otherwise it is true.

The approximate nature of the analysis results from the use of *or* in the point 4, and because in point 5 one supposes that a while loop can always terminate.

3.3 Termination of atoms

Atoms should always terminate instantly (during the instant in which they are started). As seen in 2.1, it is the programmer's responsibility to ensure that atoms do actually terminate. During atom execution, Cyclone errors[2] are trapped at the outermost level of the executing thread. There is another error that should be detected in order to get a safe language: a linked thread should never execute an atom containing an attempt to take a lock (POSIX mutex); otherwise, the thread could be blocked forever, if the lock is never released, which would actually prevent the other threads to execute.

The definition of static analysis techniques for detection of non-terminating atoms is left for future work.

4 Related Work

In [5] an extension of Cyclone to multi-threading is presented. This extension considers preemptive threads and focuses on elimination of data races. It is noted that with a shared-memory multi-processor, data races can be unsafe. Basically, in this proposal, programmers should assign each data object a lock that a thread must hold to access the data. Actually, in Loft+Cyclone, no data race can occur for systems respecting the software architecture of 2.7. The point is thus to provide a static analysis to verify that the architecture is correct. This is left for future work. Note anyway that the techniques to be used for such a verification seem very close to the ones used in [5], based on regions.

In [2] new methods are developed to statically bound the resources needed for the execution of systems of cooperative threads sharing the same instants. A system of compositional static analyses guarantees that each instant terminates. This is exactly what is needed for the FairThreads model and it is planned to study how these techniques should apply in this case. Note that [2] also describes how the size of the values computed by the system at one instant is bounded as a function of the size of its parameters at the beginning of the instant.

[2] Division by 0 does not presently (version 0.8.1) raise a Cyclone exception, but it is planned to do so.

Instantaneous termination of Esterel programs is studied in [7]. The Esterel model is based on instants and instantaneous loops are to be rejected. This is done by a static analysis close to the one presented here; however, the issue in Esterel is complicated by several specific features of the language (for example, the so-called "causality cycles", which do not arrive in Loft).

5 Conclusion

The Loft+Cyclone language is a first step toward a safe concurrent language capable to benefit from multi-processors architectures (at least SMP ones).

The language is implemented on Linux as a pre-processor of Cyclone. The execution engine is written in Cyclone. The standard `pthread` library is used for autonomous schedulers and unlinked threads. Using the multi-threaded version of Boehm's GC, the assumption was made that the Cyclone implementation is thread-safe (which seems to be the case to a large extent; anyway this is an objective of the designers of Cyclone). Under this assumption, one gets an extension of Cyclone to concurrency which is as safe as Cyclone for sequential code, with some safety verifications available for concurrent code.

Two static analyses should be added in order to achieve a fully safe concurrent language:

- Verification that all atoms executed by linked threads indeed terminate.
- Verification of the conformity to the architecture of 2.7 which excludes data races.

References

[1] Loft site: `http://www-sop.inria.fr/mimosa/rp/LOFT`.

[2] Roberto M. Amadio and Silvano Dal Zilio. Resource control for synchronous cooperative threads. In *Proc. of CONCUR 2004 – 15th International Conference on Concurrency Theory*, pages 68–82. Lecture Notes in Computer Science, Vol. 3170, Springer-Verlag, 2004.

[3] Frédéric Boussinot. Fairthreads: mixing cooperative and preemptive threads in C. *Inria research report, RR-5039*, December 2003, to appear in *Concurrency and Computation: Practice & Experience*.

[4] Frédéric Boussinot. *Reactive Programming of Cellular Automata*. Inria research report, RR-5183, May 2004.

[5] Dan Grossman. Type-safe multithreading in Cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 13–25, New York, NY, USA, 2003. ACM Press.

[6] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[7] Olivier Tardieu and Robert de Simone. Instantaneous termination in pure esterel. In *Static Analysis Symposium, San Diego, California*, 2003.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399